LEDOUX-LEVIN Simon
MARZIN Alan

Master 2 ILa – 2017/2018
ISTIC – Université Rennes 1

Rapport AOC :
Active Object / Asynchronous Observer

I - Introduction :

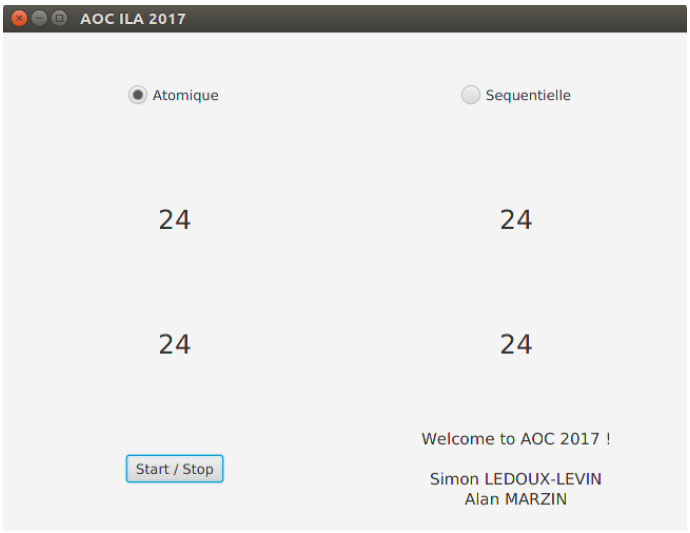
Dans le cadre du cours d'AOC en master 2 ingénierie logiciel, nous avons réalisé une application JAVA mettant en œuvre le patron de conception Active Object / Observer asynchrone.

Ce projet consistait donc a créé une entité générant des valeurs (le Générateur) et une autre les affichant appelé, l'Afficheur. Pour diffuser ces valeurs, un canal fait office de proxy entre le générateur et l'afficheur.
Le but étant de pourvoir gérer plusieurs clients a la fois de façon asynchrone grâce au Thread.
L'objectif du patron d'Active Object est d'introduire la gestion de la concurrence, en utilisant l'invocation de méthode asynchrone et un planificateur pour gérer les requêtes.

De plus, nous avons également utilisé la librairie JavaFX pour l'interface graphique. Nous avons également utilisé Log4J pour la gestion des logs.

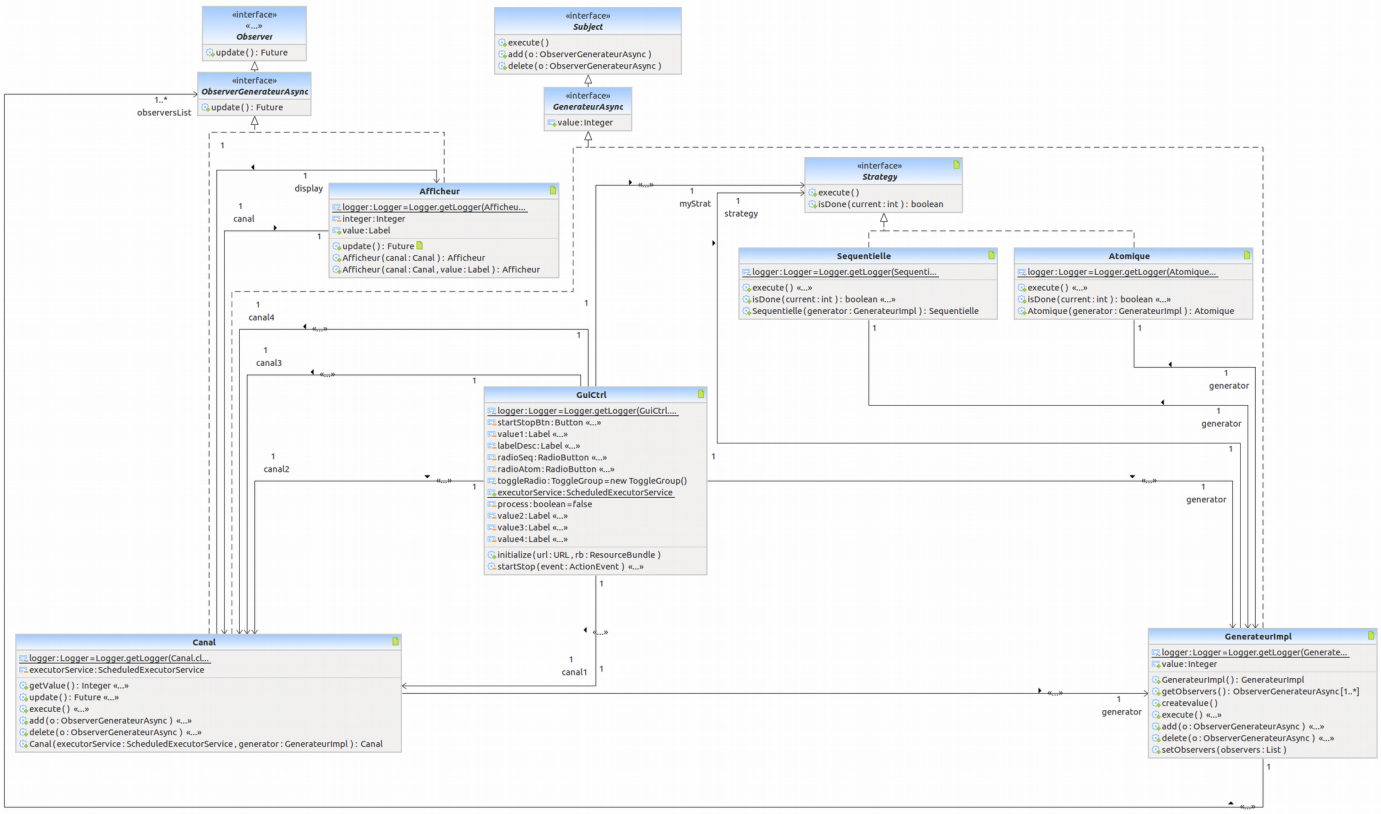
II - L'IHM :

Voici une capture de l'IHM :



Tout d'abord, nous avons le choix de l'algorithme : Atomique ou Sequentielle (cf **VI – Pattern Strategy**). Nous remarquons au centre nos 4 éléments d'affichage. Ces labels vont réceptionner la valeur générée par le générateur de façon asynchrone. En fonction de l'algorithme choisi, la mise a jour des données n'est pas la même. Et enfin, un bouton Start/Stop permet de lancer ou couper le process.

III – Architecture :



Ci-dessus, l'architecture de notre application. Nous avons 4 packages :

istic.main :

- **App.java :** Lanceur de l'application. Exécute l'IHM et initialise les éléments.
- A**fficheur.java** : Classe mettant à jour le composant graphique affichant le valeur généré. 4 objets Afficheur sont donc créés.
- **GuiCtrl.java** : Classe JavaFx de l'interface graphique. Cette classe est lié au au fichier ihm.fxml dans les ressources. Elle réceptionne et gère les actions de l'utilisateur.

istic.proxy :

- **Canal.java** : Classe faisant l'intermédiaire entre le générateur et les observateurs / clients. Un canal est créé par afficheur. Ce classe se charge de notifier aux observateurs le changement de valeur générée de façon asynchrone.

istic.observer :

- **Subject.java**:interface du sujet observé, en l'occurrence, notre générateur.
- **GenerateurAsync.java** : interface du générateur asynchrone (étend Subject).
- **GenerateurImpl.java** : Implémentation du générateur asynchrone.
- **Observer.java** : interface des observateurs
- **ObserverGenerateurAsync.java** interface des observateurs du générateur asynchrone (étend l'interface Observer) qui est en l'occurrence, Afficheur.

istic.strategy :

- **Strategy.java :** Interface des algorithmes pour le design pattern Strategy.
- **Sequentielle.java** : Implémentation d'un algorithme de Strategy.
- **Atomique.java :** Implémentation d'un algorithme de Strategy.

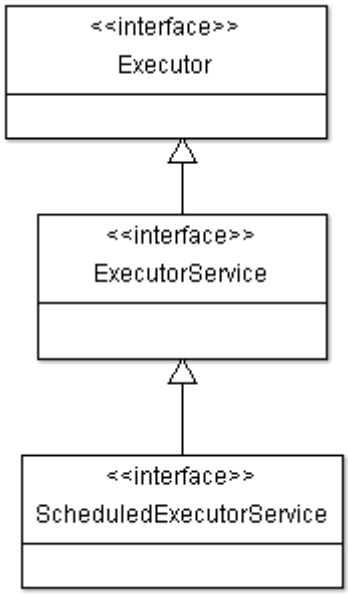
IV – Pattern Observer :

Lors de ce projet, nous avons dû implémenter le design pattern observer.

Ce patron est utile pour envoyer un signal à des modules qui jouent le rôle d'observateurs. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent.

Le DP observer permet de notifier les changements de valeur à l'observateur, en l'occurrence l'Afficheur par le Canal.

V - Traitement asynchrone :



Pour le traitement asynchrone nous avons utilisé le framework Executor. Ce framework intègre une classe nommé ScheduledThreadPoolExecutor.
Cette classe permet de lancer des Thread (nombre fixe) à intervalle régulier.

VI – Pattern Strategy :

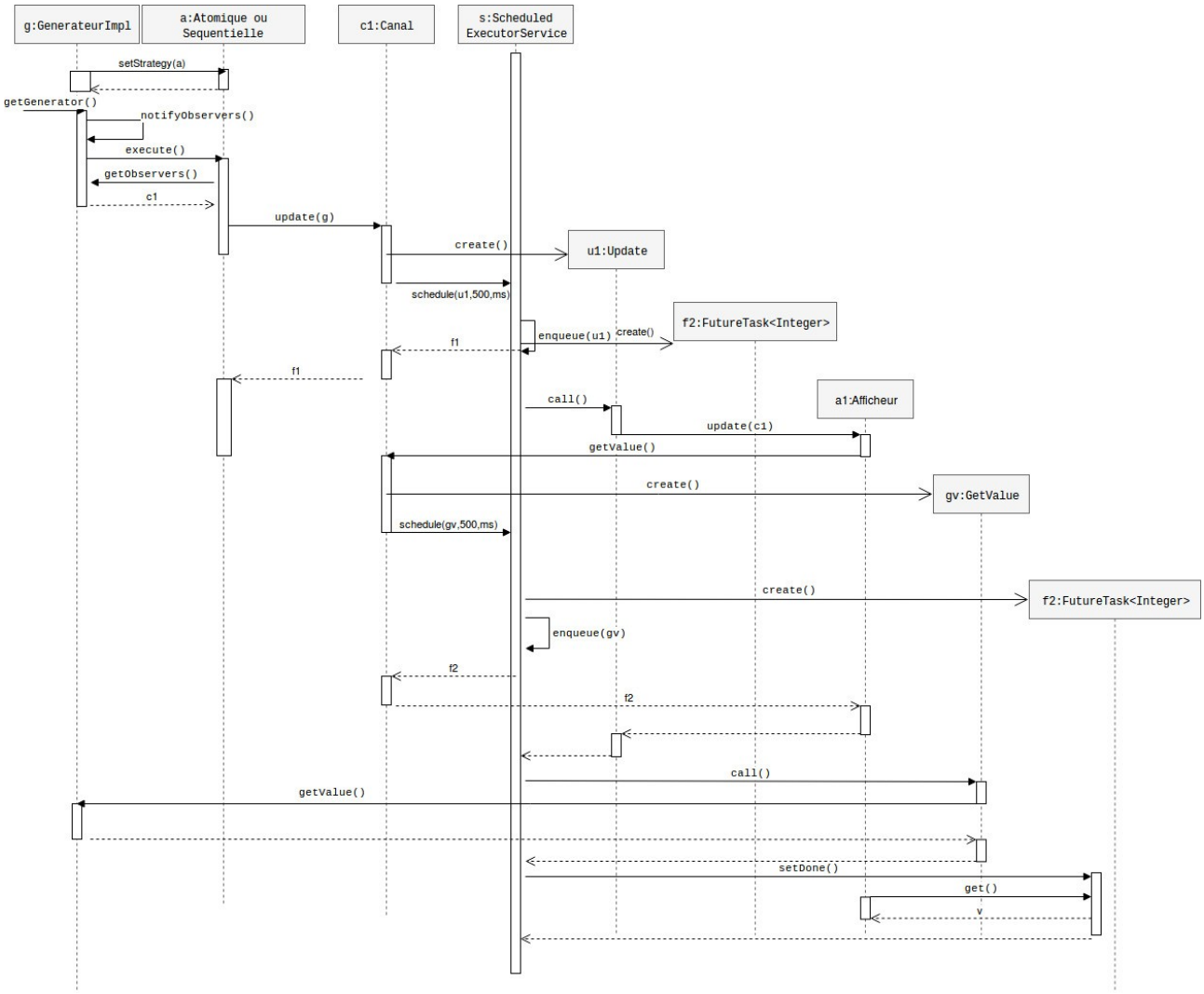
Le patron stratégie est un patron de conception de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée pendant l'exécution du programme
Le patron de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application.
Dans notre cas, nous avons utilisé ce patron pour l'algorithme de diffusion.

Nous avons deux algorithmes de diffusion :

Atomique : L'ordre d'arrivé de la demande n'est pas pris en compte.
Sequentielle : Ordre strict. Les afficheurs reçoivent leur valeurs une à une à la suite.

Cet algorithme peut être changé à la volée (dans l'IHM).

VII – Diagramme de séquence :



setStrategy détermine le type d'algorithme de diffusion (Atomique ou Séquentielle).

Conclusion :

Nous avons eu des difficulté à mettre en place l'asynchronisme dans ce projet car méconnu de notre domaine de compétence. Les cours de GLI nous ont aussi permis de pouvoir appréhender facilement l'utilisation de JavaFx et le contrôle d'IHM dans le projet.
Par manque de temps, nous n'avons pas pu aboutir le projet jusqu'au bout. Certains fonctionnalités n'ont pas été implémentées tel que l'algorithme de diffusion : **Par Epoques**.
Pour conclure, il a été très intéressant de travailler sur ce projet d'AOC. Il nous a permis de mieux comprendre les utilisations de plusieurs design pattern dans un même projet qui, certes, peut complexifier l'architecture mais offrent de multiples avantages.